Bolt Beranek and Newman Inc.

(12)

AD A112956

Report No. 4927

Development of a Voice Funnel System

Quarterly Technical Report No. 13
1 August 1981—31 October 1981

March 1982

Prepared for:
Defense Advanced Research Projects Agency

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A112 956	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Development of a Voice Funnel System Quarterly Technical Report No. 13		5. TYPE OF REPORT & PERIOD COVERED Quarterly Technical 1 August 81-31 October 81
		6. PERFORMING ORG. REPORT NUMBER 4927
7. AUTHOR(s) R. D. Rettberg		8. CONTRACT OR GRANT NUMBER(s) MDA903-78-C-0356
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman Inc. 10 Moulton Street Cambridge, MA 02238		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 3653
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE March 1982
		13. NUMBER OF PAGES 14
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Voice Funnel, Digitized Speech, Packet Switching, Butterfly Switch, Multiprocessor

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This ~~Quarterly Technical~~ Report covers work performed during the period noted on the development of a high-speed interface, called a Voice Funnel, between digitized speech streams and a packet-switching communications network.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

Report No. 4927 Bolt Beranek and Newman Inc.

DEVELOPMENT OF A VOICE FUNNEL SYSTEM

QUARTERLY TECHNICAL REPORT NO. 13
1 August 1981 to 31 October 1981

March 1982

This research was sponsored by the
Defense Advanced Research Projects
Agency under ARPA Order No.: 3653
Contract No.: MDA903-78-C-0356
Monitored by DARPA/IPTO
Effective date of contract: 1 September 1978
Contract expiration date: 31 December 1981
Principal investigator: R. D. Rettberg

Prepared for:

Dr. Robert E. Kahn, Director
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209

Table of Contents

1. Introduction

This Quarterly Technical Report, Number 13, describes aspects of our work performed under Contract No. MDA903-78-C-0356 during the period from 1 August 1981 to 31 October 1981. This is the thirteenth in a series of Quarterly Technical Reports on the design of a packet speech concentrator, the Voice Funnel.

This report describes the exception handling conventions that are supported in the programming environment of the Voice Funnel.

2. Exception Handling Mechanisms

This section discusses an important unifying concept in the Chrysalis operating environment, a mechanism for uniform reporting and handling of exceptional conditions. The mechanism resembles the exception handling mechanism in Ada, but has been modified to be compatible with C, and to meet the needs of the Voice Funnel project.

The software for the Voice Funnel is being written in the language C which was developed and used for the UNIX * Operating System. This has led us to provide a programming environment that is in many ways similar to UNIX. In UNIX, when programs need operating system services they simply call the appropriate function as a subroutine from C, using normal argument passing conventions. In our view, this is an effective approach, and we have included special hardware features in the Butterfly to improve its efficiency. Chrysalis is simply a large subroutine package, resident in each Processor Node, which is mapped into and shared among all processes on that node. Subroutines in Chrysalis can enter Kernel Mode to manipulate protected resources, but are otherwise identical to ordinary subroutines.

However, there are several areas in the UNIX architecture in which we felt compelled to make significant changes. One of these was the mechanism which allows a process to wait for one or

* UNIX is a trademark of Bell Laboratories.

more external events to occur. This will be discussed in more detail in a subsequent report. Often our concerns have centered on issues which are particularly important in a real-time multiprocessor, such as inter-process communications, scheduling efficiency, and graceful recovery from exceptional conditions. It has been our goal to write as much of our code as possible using a standard, unmodified C compiler. However, it is sometimes difficult to separate issues of operating system design from those of language design; the standard C macro preprocessor has enabled us to extend the C language itself in a few selected areas and provide solutions to some of the problems we have identified.

In our view, one of the principal weaknesses of the C language is its inability to handle exceptions gracefully. This was particularly a problem in the design of Chrysalis, since operating system functions frequently detect errors of various sorts, which must be reported to the application program. In UNIX, at least three different techniques have been used to report exceptional conditions:

1. Many UNIX functions return a special value in case of error, such as zero or minus one. Sometimes the only purpose of the value returned is to indicate success versus failure; in other cases this error indication is multiplexed with an ordinary value. For example, many UNIX functions return either a pointer to their result, if successful, or zero in case of error. To use these functions one must test the result, and take special action in case of error. Many UNIX functions record an error code in a special static variable to allow the application to determine the nature of the error.

2. UNIX also provides a "signaling" mechanism for reporting certain conditions (usually more serious errors, such as invalid memory references). When such a condition is signaled, the application program state is pushed, and control is passed either to a system-default signal handler (which would normally terminate the application process), or to a subroutine explicitly designated by the application program. This subroutine can take any action it sees fit, including cleanup, problem correction, etc., and can (sometimes) pop the saved program state and continue, terminate the application process, etc.

3. The C library includes a "setjmp, longjmp" package, which allows a subroutine to save its stack environment, then call lower-level routines. In case of error these routines can call "longjmp" to restore the stack environment and return directly to the higher-level subroutine without going through any intermediate levels. UNIX does not use this package directly to report errors, and the package as implemented in the C library is too simple to be very useful. A number of other languages such as Ada provide more sophisticated features of this type.

In the course of designing Chrysalis we gave the problem of error reporting considerable thought. We concluded, for reasons discussed below, that the methods used by UNIX are frequently inappropriate and tend to encourage poor programming practices, such as failing to test for error conditions which are reported by the operating system but which the programmer did not expect to occur.

As an example, in UNIX, the system call that writes to the disk returns an error code if the disk is full and the write failed to happen. Many application programs do not bother to handle this return code and as a result the operation of the system is unreliable when the disk is nearly full. The mistake was to require a special test in every program for a failure that should rarely if ever happen. A simple default handler should have been provided instead.

We have chosen to adopt instead a mechanism which is a generalization of the "setjmp" package, and provides features somewhat like the exception handling features in Ada. Using terminology taken in part from MACLISP, we call this the "catch/throw" facility.

2.1 Catch and Throw

In Chrysalis, when an exceptional condition is detected, the low-level routine which detects it calls the function "throw" with arguments which describe the nature of the problem. "Throw" scans up the stack until it either locates a "handler" for the condition, or until it reaches the top of the stack. If it reaches the top of the stack, it invokes a default handler which records the nature of the problem for the programmer and terminates the process. For example:

```
get_number(string)
{
  ...
  if (error)
    throw(CONSISTENCY, "get_number: syntax error", string);
  ...
  return number;
}
```

If the error flag is set, this throw "raises" the CONSISTENCY exception, which indicates that its input arguments were inconsistent or invalid. In addition, it returns both an error message and a pointer to the faulty argument.

When a handler is invoked, the applications program has the ability to analyze the nature of the problem which caused the throw, and to deal with it in one of three ways:

1. It can decide to correct, report, or ignore the problem, and continue processing at the level at which the handler is running.
2. It can abort the specific task it was attempting to do, for example by returning the resources it was using, unlocking locks, etc., and then call "rethrow" to pass the problem up to a still higher level.
3. It can abort, and "throw" a different, more appropriate error, or even terminate the process.

This mechanism solves the problems involved in checking every function call for the special value which would indicate a problem, and then passing a special value back in turn. It also provides a uniform mechanism for passing back messages which describe the problem. The mechanism can be used equally well to report problems detected by Chrysalis, by ordinary library routines, or by application programs. The fact that this mechanism does not save the program state is an advantage over the UNIX signalling mechanism in those cases where it is inappropriate to return to the interrupted program.

2.2 Impact on UNIX Compatibility

Although this mechanism is new in Chrysalis, it is not as incompatible with standard UNIX as one might at first suppose. There are several reasons for this. First, UNIX programs written to return and check for special codes could be imported to Chrysalis without being modified to use catch and throw. Second, although the signalling mechanism is not available in Chrysalis in a compatible form, most UNIX programs do not make extensive use of it anyway. Third, the setjmp package is even less used, and in any event, programs which use it can either continue to do so, or be easily converted to use the catch/throw package instead.

The signalling mechanism in UNIX is used for a number of quite different purposes. It appears that the UNIX designers thought of signals as a mechanism which would provide solutions to a wide range of operating system problems. We feel that this was an error, and that UNIX-like signalling mechanisms should rarely if ever be relied upon. Perhaps fortunately, the UNIX implementation of signals had a number of serious bugs and shortcomings which have only recently been corrected. Because of these, most existing programs avoid the use of signals whenever possible. UNIX signals can be divided into classes as follows:

- Machine exceptions.

  These vary from machine to machine, but generally include illegal instructions, illegal memory accesses, and divide by zero. Generally it is impossible to return from such an exception, and would be bad programming practice even if possible. In Chrysalis these conditions trigger standard system-originated throws, which include location and error code information. The catch/throw mechanism is ideal for handling such conditions.

- Serious errors in calls to UNIX functions.

  Some errors, such as "bad argument to system call" and "write on a pipe with no one to read it", cause signals, since ignoring them is not a reasonable thing to do. The catch/throw mechanism is ideal for handling errors of this type.

- External requests.

  These include signals which are normally triggered by the timer ("alarm clock"), the tty input handler ("dataset has hung up", "user typed the 'interrupt' key", "user typed the 'quit' key"), or by some other process ("kill this process immediately", "associated process has terminated"). This group of signals seems quite natural at first glance, but can cause serious technical problems. Chrysalis generally handles this sort of occurrence as an Event; in addition, a special mechanism is available to immediately kill an errant process.

Unfortunately, the obvious handlers for most external request signals have not-so-obvious bugs which are hard to fix or even to avoid at all. The underlying problem is that the state of the main program is arbitrary and unknown at the time the signal handler is entered. Data structures may be in the process of being updated; key variables may be momentarally invalid; inter-process locks may be held; formatted tty output may be in progress. If these concerns are not to be ignored, there is little the typical signal handler can validly do but to set a flag which the main program can test at a convenient time when its state is clearcut. This is exactly what the Event mechanism is designed to do, without the need to invoke a signal handler at all. Even if the goal of the signal is to terminate the signalled process as quickly as possible, it may be important that outstanding shared resources such as locks be released cleanly, and perhaps that other related processes be informed in turn.

In an environment with catch and throw it might seem natural to have a signal handler initiate a throw to trigger the appropriate cleanup mechanisms. This approach, which is specified in Ada, is extremely difficult to implement correctly. The difficulty arises when an external signal attempts a throw just as the main program is acquiring or releasing some shared resource. At all such points the main program must be coded in such a way that its catch handler can tell exactly what has been done so far and how to abort or complete the critical operation. In many cases the only practical approach would be to disable all signals in such critical regions; but this is expensive, clumsy, and error-prone.

In Chrysalis we have avoided such problems by outlawing signals. The application program establishes an Event to be posted, instead of establishing a signal handler. Then it checks that Event at an appropriate point or points in the main program, and calls throw "voluntarily" if the Event has been posted.

2.3 Comparison with Ada

Chapter 11 of the Ada reference manual (July 1980) specifies the Ada exception handling mechanism. While we have serious reservations about Ada's primitives for intertask communications, we basically like Ada's exception handling mechanism, and the catch/throw package can be viewed as our attempt to implement this idea in a reasonable way. We have avoided using similar notation to minimize confusion for those familar with Ada, since our mechanism is by no means identical. Most of the differences are cosmetic, but some are more fundamental and deserve further discussion.

- Data passed when an exception is raised.

  Ada passes no data, beyond the simple fact that a certain named exception has occurred. In our experience, many exceptions are unexpected and reflect bugs of some sort. Thus it is important to pass back all the information available at the time the exception was raised for debugging purposes. Even if an exception is expected occasionally, it is sometimes valuable to pass back a result or error code.

- Debugging exceptions.

  It can be valuable to treat certain classes of exception as breakpoints in interactive environments. A backtrace at the time of an unexpected exception can be extremely useful. On the other hand, certain kinds of exception are "normal" and must not be handled this way. Exceptions which remained unhandled in the main procedure should go to a "default" handler to capture debugging information before the task is terminated.

- The Ada FAILURE exception.

  Handling this exception may easily lead to the type of problem described above for external request signals, since there is no guarantee that the receiving task is in a state which can be interrupted.

2.4 Using Catch and Throw

One must think of the catch/throw facility in terms of C's stack-based model of program execution. A "catch" is a special marker which is placed onto the run-time stack by a procedure, so that when that procedure returns, the catch is popped off of the stack along with the rest of the procedure's stack frame. "Throw" is a special procedure which will pop frames off of the stack until it finds a catch, and then hand control over to the procedure which installed the catch. One can think of throw as sort of a "super-return" statement, which allows one to return to a procedure which is an arbitrary distance back up the stack.

The throw call takes three arguments: an exception code, an arbitrary text string, and an optional arbitrary value (a long). Exception codes have two parts, an exception class and an exception number. Exception classes are defined by the system, but some classes are reserved for application program use. There are at most 24 exception classes, which are bit-encoded in the high-order 24 bits of a long integer. The low order 8 bits specify the exception number within a specific class. System-defined exceptions always have numbers greater than 128, while user-defined exceptions have numbers from 0 to 127. The text string is used to identify the particular exception being thrown;

the arbitrary long can be used to return a number or pointer, either as a result or as a debugging aid.

The syntax for using a catch is as follows:

```
catch
  maincode;

onthrow
  when (throwcode == CODE1)
    code1;
  when ((throwcode & CLASS2) != 0)
    code2;
    rethrow;
  when (strcmp(throwtext, "Exception 3"))
    code3;
endcatch
```

If any exception occurs during the execution of "maincode", the "onthrow" clause is evaluated. Then, if none of the "when" clauses is true, or if the "rethrow" statement is executed, control will pass through the catch (presumably to be caught further up the stack). This example will catch a throw of either CODE1, CLASS2, or "Exception 3" and pass any other throws.

Because the catch/throw package is implemented using preprocessor macros, certain restrictions were unavoidable. This approach was much simpler than modifying the C compiler, and more flexible as well. Generally, the restrictions do not impose much of a burden on the programmer. Unfortunately, however, violation of these restrictions is generally not detected by the compiler, and simply produces erroneous code. These are the restrictions:

- "maincode" may not contain any "gotos", "breaks", or "continues" which branch out of "maincode", and nothing may branch into "maincode".

- "return expr;" statements must be written as "RETURN expr;", which will properly adjust the catch/throw stack BEFORE the expr, if any, is evaluated. RETURN expands into more than one C statement, so use brackets where appropriate. For example, brackets are required in the statement "if (conditional) { RETURN 6; }".
- When an onthrow clause is entered, all register variables are restored to their values at the time the catch block was entered. If a local variable is modified inside of a "catch" and its value needs to be maintained after a throw, it must not be declared as a register variable.

Within an "onthrow" clause, the exception code being processed is available as "throwcode". A pointer to the text string which was passed to throw is available as "throwtext", and the value is available as "throwvalue"; a pointer to the call to "throw" is available as "throwlocation". Specifying "when (TRUE)" in an "onthrow" clause will catch any and all throws not already caught by earlier "when" clauses. After performing various cleanup actions, the catching routine can propagate the same exception with the "rethrow" command.

This mechanism does not allow throws from other processes or from asynchronous operating system functions such as timers. To handle such cases, the user should establish an Event, and poll or explicitly wait for it to be posted. This mechanism could be used to handle fatal errors such as protection violations, as long as they cannot occur asynchronously. For example, if you wait for a lock, then enter a catch block which unlocks the lock, a protection violation between the time you obtained the lock and the time the catch was established would cause trouble. However, since these should both be well-debugged system procedures, they can be counted on not to produce protection violations.

The catch/throw mechanism is relatively efficient. As currently implemented, the biggest cost is that each time a "catch" is entered, a copy is made of all the MC68000 registers. When a throw is caught, all registers are reloaded from this copy before entering the onthrow clause. An alternative implementation would be to restore all temporary registers used by each nested subroutine as we proceed up the stack. This would avoid the overhead of explicitly saving the registers in the "catch" code, and would have the nice feature of preserving changes to register variables made inside the catch block. Unfortunately, it seems that this approach would be impossible without making significant changes to the C compiler itself.

A flaw in current implementation of the throw mechanism is that it provides an obscure hole in the protection mechanism. Closing the hole would involve the extra run-time expense of dynamically allocating catch control blocks in segment F8 instead of on the application's pushdown list. We do not consider this expense worthwhile for the Voice Funnel.

DISTRIBUTION OF THIS REPORT

Defense Advanced Research Projects Agency
Dr. Robert E. Kahn (2)
Dr. Vinton Cerf (1)

Defense Supply Service -- Washington
Jane D. Hensley (1)

Defense Documentation Center (12)

USC/ISI
Dr. Danny Cohen (2)

MIT/Lincoln Labs
Clifford J. Weinstein (3)

SRI International
Earl Craighill (1)

Rome Air Development Center/RBES
Neil Marples (1)

Defense Communications Agency
Gino Coviello (1)

Bolt Beranek and Newman Inc.
Library
Library, Canoga Park Office
R. Bressler
R. Brooks
P. Carvey
P. Castleman
G. Falk
F. Heart
M. Hoffman
M. Kraley
W. Mann
J. Pershing
R. Rettberg
E. Starr
E. Wolf